International Journal of Software Engineering
and Knowledge Engineering
Vol. 24, No. 10 (2014) 1489–1508

DOI: 10.1142/S0218194014400166

World Scientific
www.worldscientific.com

Characterizing the Problem of Developers' Assignment for Merging Branches

Catarina Costa[*,†,‡], J. J. C. Figueiredo[*,§], Gleiph Ghiotto[†,¶] and Leonardo Murta[†,‖]

[*]*Technology and Exacts Science Center, Federal University of Acre — UFAC*
Rio Branco, Acre, Brazil

[†]*Computing Institute, Fluminense Federal University — UFF*
Niteroi, Rio de Janeiro, Brazil
[‡]*catarina@ufac.br*
[§]*jairfigueiredo@ufac.br*
[¶]*gmenezes@ic.uff.br*
[‖]*leomurta@ic.uff.br*

During the software development process, artifacts are constructed and manipulated by many developers working in parallel. A common practice to manage parallel development is the use of branches in the version control system. Usually, at some point, the merge of these branches may be necessary. This process can combine two independent and eventually long sequences of commits, which may have been performed by different developers. Conflicts resulting from the merge of parallel changes may arise. When these conflicts are not automatically solved by the version control system, the developers in charge of the merge process must act. Normally, the developers' knowledge regarding the changes performed in parallel is usually not taken into consideration when assigning developers to the merge task. With this in mind, the goal of this work is to characterize the problem of developers' assignment for merging branches. To do so, this work analyzed merge profiles of eight software projects and check if the development history is an appropriate source of information for identifying the key participants for collaborative merge. In addition, this work presents a survey on developers about what actions they take when they need to merge branches, and especially when a conflict arises during the merge.

Keywords: Merge conflicts; collaborative merge; developers assignment.

1. Introduction

It is common to have software development teams coding in a collaborative way, where different developers work in parallel, changing the same files, and, sometimes, changing the same parts of the code inside a file. Those developers may be working in different branches to evolve the source code in an isolated way (e.g. isolating a corrective maintenance from an adaptive one). However, usually, in some moment those changes need to be combined through a merge process.

In this process, merge conflicts are common, especially in large and distributed software projects [1]. Identifying and resolving these conflicts are not trivial tasks. While two simple concurrently changes might be individually correct, their combination may become inconsistent. These conflicts demand conciliating choices and, consequently, this whole scenario usually introduces rework [2].

Most contemporary version control systems adopt optimistic version control, and the last developer to commit is in charge of merging his/her workspace into the repository. Most of the times, he/she makes the merge decisions alone. However, when branches should be merged, although it is possible to select any developer to perform such task, no explicit support is provided for this decision. Usually, the decision of which developers are going to perform branch merging is not based on the knowledge related to the work nor if they are actually the best developers to perform such task.

Some tools, such as KDiff[a], WinMerge[b], Diffuse[c], and SourceGear DiffMerge[d], feature automated solutions for merge conflicts, but they are not able to resolve conflicts that occur in the same region of the files. In contrast, CASI [3], CloudStudio [4], CoDesign [5], Crystal [6], Palantír [2], SafeCommit [7], Syde [8], and WeCode [9] try to avoid conflicts through awareness. Their main goal is to alert developers in a proactive way of situations that may lead to conflicts [4, 10]. However, these approaches require certain conditions, such as developers working in sync in terms of time and over the same branch, which is not always the case for distributed software development. All in all, even adopting the aforementioned tools, conflicts may still exist. Finding the appropriate developers to solve them is an important challenge.

The use of the existing shared knowledge of software projects is a missing element for conflict resolution [1]. The configuration management repositories have a trail of who changed the software and can help in identifying the most appropriate developers for solving conflicts. This information could be extracted, for instance, from the project development history until the point where the merge takes place.

This paper provides an analysis of how parallel development occurs using branches and how branch merging is usually performed. To do so, two complementary studies were performed. In the first study we analyzed the merge profiles of eight software projects versioned using the Git[e]. This post-hoc study was guided by the following for research questions, whose objective is the understanding of how branch merging occurs.

- Q1: What is the average number of developers per branch?
- Q2: What is the average number of commits per developer in each project?

[a] http://kdiff3.sourceforge.net/
[b] http://winmerge.org/
[c] http://diffuse.sourceforge.net/
[d] https://sourcegear.com/diffmerge/
[e] http://git-scm.com

- Q3: What is the maximum number of developers in each branch?
- Q4: How many merges have the same developers in both branches?

The second study is based on a survey where developers answered questions about what actions are taken by them when they need to merge branches, and especially when a conflict arises during the merge. The developers' opinion is considered important to know how they deal with conflicts in real projects and to identify if they actually need support to solve merge conflicts.

The answers to both analyses provide important insights on how much the project development history can assist in the designation of key participants to resolve merge conflicts. This information can help in defining new methods for assign developers to participate in merge sessions, known as collaborative merge.

In a previous conference paper [11] we presented the post-hoc analysis, looking at the characteristics of merge in eight software projects. This work extends our previous work by introducing the second analysis, surveying 164 developers about how they merge parallel work. These two perspectives are complementary and fundamental to understand the socio-technical aspects related to branch merging.

This paper is organized in seven sections, including this introductory section. Section 2 presents the concepts involved in the research. Section 3 presents the related work. Section 4 presents the research method we adopted for both the post-hoc analysis and the survey. Section 5 shows the post-hoc analysis results and some insights on how the development history could help in the identification of merge participants. Section 6 shows the analysis of the survey's answers. Section 7 discusses how the post-hoc and survey analyses can help in the assignment of developers to merge branches in three complex scenarios. Finally, Sec. 8 concludes the paper presenting some future work.

2. Background

According to Mens [12], three types of conflicts may arise during merge: textual, syntactic, and semantic. Textual conflicts, also known as physical conflicts, occur when concurrent operations (e.g. addition, removal, or edition) take place over the same physical parts of a file (e.g. same line). Syntactic conflicts occur when concurrent operations break the syntactic structure of the file when combined. The syntactic structure of a file is usually defined in terms of a schema or grammar. Finally, semantic conflicts occur when concurrent operations break the semantics of the file when combined. The semantics of the file can be expressed both in terms of the programming language semantics or expected program behavior (usually verified by test cases).

The existing version control systems are limited to textual conflicts. This way, a conflict arises when two or more developers make inconsistent changes in the same part of the code. When this happens, the changes cannot be incorporated into the repository until the conflict is resolved by some developer [6]. Figure 1 shows an

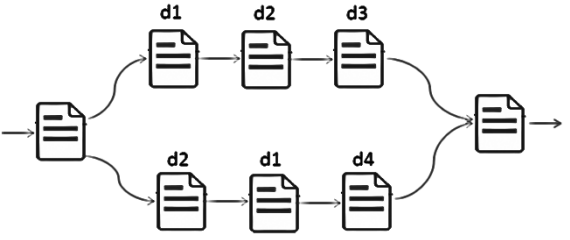

Fig. 1. A software merge scenario.

example of merge, where four developers (d1, d2, d3, and d4) work in two branches, being d1, d2, and d3 in the first branch, and d2, d1, and d4 in the second branch. In a given moment, d4 decides to merge the second branch into the first branch. In this case, in most version control systems, d4 must conciliate the choices and possibly edit some parts of the code written by the other developers.

This example shows that, disregarding the content contribution of each developer, d1 and d2 probably know how to perform the branch merging better than d3 and d4 for this specific situation. This occurs because they worked in both branches and, consequently, understand better the changes performed in both branches. The developer d4 could be at least assisted by d1 and/or d2 to more effectively conciliate the decisions.

3. Related Work

The majority of the works in the literature on merge conflicts focus on the prevention of conflicts through awareness. By monitoring workspaces, awareness tools, such as Palantír [2] and CloudStudio [4], notify the developers every time a change that may lead to a conflict is being made. Other approaches, such as Crystal [6] and WeCode [9], work by continuously trying to merge the local copies of the various users. Although these works have distinct approaches, conflict prevention is their main goal.

All approaches presented here require developers to be constantly synced in time throughout the development of the software, but this is not always possible in distributed software development. As mentioned before, these works also expect that developers are working over the same branch. However, for different reasons [13], sub-teams work in parallel over different branches. Thus, conflicts cannot be preemptively detected, which would imply solving them during merge anyway.

Assuming that conflicts can always occur, Koegel *et al.* [14] present a collaborative model merging approach that shows conflicts as open questions. To do so, the approach allows to represent conflicts explicitly as parts of the model and thereby to postpone them. This facilitates the discussions on conflicts and the collaborative conflict resolution. Their study shows that developers like to discuss conflicts in a

considerable percentage of time. The authors defend that the resolution of a conflict must not be assigned to only one developer, becoming a collaborative task. Complementarily, Nieminen [1] proposes a web-based tool (CoRED) that allows multiple users to resolve merge conflicts in a collaborative way. When facing a conflict, developers can invite a workmate to take part in a conflict resolution session, aiming at reaching a consensus. In this work, the authors focus on conflict resolution with the assistance of a tool.

The last two works discussed in this section contributed interesting insights for this paper. Our work differs from them because we aim at analyzing merge profiles and how developers perform merge, and check if the development history is an appropriate source of information for identifying the key participants for collaborative merge. It is important to notice that both recognize the importance of having the correct developers during merge, but none of them provide insights on how to select such developers.

4. Materials and Methods

Our research followed the process depicted in Fig. 2 for comprehending how merge happens in practice, analyzing the artifacts of projects and the opinion of developers. This process comprises the project selection, a post-hoc analysis, a survey, and the consolidation of results.

In the project selection step, eight projects were chosen: Django, Git, JQuery, MaNGOS, Perl5, Rails, Voldemort, and Zotero. These projects were selected because they have different sizes, are popular, are coded in different programming languages, have different goals, and are very active. This contributes for a more embracing analysis. After selecting these projects, they were cloned from Github[f], being locally available for the next step of our research process.

In the Post-hoc Analysis Step, the history of each selected project was analyzed to extract information about developers, commits, and merges. To do so, we implemented an automated infrastructure that parses the history provided by Git and extracts all

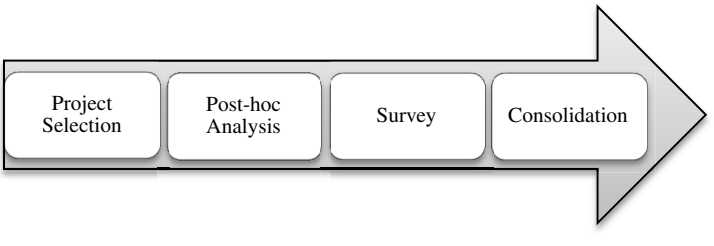

Fig. 2. Research steps.

[f]https://github.com/

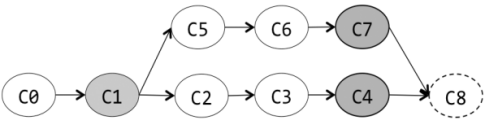

Fig. 3. A simple merge example.

merge commits. Moreover, it is able to identify the parent commits that were merged and navigate until the common ancestor just before forking the history.

Figure 3 shows an example of merge commit (C8), two parent commits that were merged (C4 and C7), and the common ancestor (C1). From these commits, it is possible to identify the commits within each branch. These commits are comprised between the common ancestor and each one of the parent commits that were merged (including the parent commits). The first branch has three commits (C5, C6 and C7) and the second branch has also three commits (C2, C3 and C4).

After identifying each commit that was merged, we extracted the author information (name and e-mail). Next, we computed multiple information, such as: who committed in each branch, how many commits they did, who committed in both branches, among others.

The most complex merges in terms of developers were prioritized by inverse sorting the merges by the harmonic mean of the number of developers in each branch. An in-depth analysis was carried out for some of the most complex merge scenarios of these projects in order to understand which developer could participate in a collaborative merge section.

We complemented the Post-hoc Analysis with a Survey, where we analyzed the developers' opinions with the goal of collecting information about how merge conflicts are solved in real projects. The steps followed to run this survey were based on Pfleeger and Kitchenham [15] work, which consists in: setting specific and measurable objectives; planning and scheduling the survey; ensuring that appropriate resources are available; designing the survey; preparing the data collection instrument; validating the instrument; selecting participants; administering and scoring the instrument; analyzing the data; and reporting the results.

The survey's questionnaire was constructed in three rounds. In the first round, an initial questionnaire was built. This questionnaire was divided into three sections: (1) basic information (2) professional experience, and (3) experience with version control systems. The third section has questions related to policies to avoid conflicts, policies to solve conflicts, and so on. In the second round, other researchers reviewed the questionnaire and some changes were performed. Finally, in the last round, we performed a pilot of the survey with three software development practitioners aiming at validating the questionnaire. Based on the answer and critics, we adjusted and improved the questionnaire once again. The pilot answers were discarded and the final version of the questionnaire was used in the survey.

The questionnaire was sent to developers by email, together with some contextual information like: objective of the research, expected background, and the estimated time to answer (5 minutes). Open and closed questions were used in the survey, summing up a total of 16 questions, where 7 questions were closed-ended questions. The closed-ended questions were composed of multiple choices and Likert-scale questions.

Finally, the consolidation step comprises answering the questions presented in Sec. 1 and analyzing the survey responses. In Sec. 5, the results were organized in a way that shows the merge profiles and information on the most suitable developers to take part in conflict resolution, answering the research questions. In Sec. 6, the results and analysis of the survey are presented. Both analyses, technical (post-hoc) and social (survey), expected to understand better the factors that may influence the allocation of participants in collaborative merge.

5. Discussion About the Post-hoc Analysis

This section presents the answers to the research questions posed in Sec. 1, according to the research process described in Sec. 4. Besides getting merge behavior patterns, information such as the average number of commits by developer per branch, and the number of developers working in branches, amongst other information, are also presented. Table 1 describes the analyzed projects with their total number of merges and developers. These data were extracted by the end of 2013.

By analyzing the information shown in Table 1, it is noticeable that the projects range from just few hundreds of merges (Zotero, JQuery, and MaNGOS) to thousands of merges (Perl5, Git and Rails). It is also observable that the projects have varied numbers of participants, ranging from tens in Voldemort and Zotero to thousands in the Git and Rails. This diversity is important to allow us exploring different situations, enabling a wider observation over the development projects. In the following, we answer each research question presented in Sec. 1.

Table 1. Project's general description.

	General information		
Project	Purpose	# Merges	# Developers
Django	Python web framework	404	396
Git	Distributed revision control system	7434	1103
JQuery	JavaScript Library	250	133
MaNGOS	A full-featured MMO server suite	255	266
Perl5	Programming language	1607	789
Rails	Ruby web framework	6147	2528
Voldemort	Distributed database	406	59
Zotero	Citation manager	223	16

5.1. *Q1: What is the average number of developers per branch?*

The first question observes the number of developers involved in each branch of the merges, for each project.

The branches of all the merges, which comprise commits between each parent of the merge and the common ancestor, were processed and the results are shown in Table 2. It is important to note that there is a reason to number the branches: "the first parent is the branch you were on when you merged, and the second is the commit on the branch that you merged in" [16].

Table 2. Average number of developers per branch.

	Developers per branch			
Project	Average branch 1	Standard deviation	Average branch 2	Standard deviation
Django	7.05	20.60	2.64	8.88
Git	24.00	53.63	1.99	14.01
JQuery	4.56	7.26	1.57	2.07
MaNGOS	1.58	1.86	5.00	4.88
Perl5	2.02	7.01	3.46	6.51
Rails	10.95	43.77	2.41	8.58
Voldemort	1.47	1.67	1.73	1.45
Zotero	1.38	1.16	1.56	0.94

It is observable that, for example, Git has an average of 24 developers per merge on branch 1 and only 1.99 developers on branch 2. The same occurs in other three projects, having high values on branch 1: Django, JQuery, and Rails. In contrast, MaNGOS, Perl5, Voldemort, and Zotero have a higher average of developers on branch 2. However, with a clear exception of MaNGOS, the difference is not significant. The discrepancy of developers in branch 1 and branch 2 is probably an effect of the branching strategy [12] adopted in each project.

5.2. *Q2: What is the average number of commits per developer in each project?*

The second question observes the average number of commits per developer. To get the numbers in Table 3, the totality of commits was divided by the totality of project participants.

The number of commits per developer shows the average participation of the developers in the project. Furthermore, it allows for the identification of the developers that participate in the most and, therefore, being a possible candidate to resolve merge conflicts.

5.3. *Q3: What is the maximum number of developers in each branch?*

The third research question observes the maximum number of developers in each of the merging branches. Table 4 shows the merges with the highest harmonic mean of

Table 3. Average commits per developer.

	Commits per developer	
Projects name	Average	Standard deviation
Django	10.88	17.49
Git	176.26	418.56
JQuery	12.52	24.08
MaNGOS	7.31	21.37
Perl5	12.17	50.25
Rails	33.51	100.31
Voldemort	23.08	43.12
Zotero	42.06	74.08

Table 4. Maximum number of developers per branch.

	Maximum number of developers			
Projects name	Merge commit hash	Branch 1	Branch 2	Harmonic mean
Django	9cc6cfc4057e...	8	12	9.60
Git	a8816e7bab03...	158	10	18.80
JQuery	63aaff590ccc...	20	3	5.22
MaNGOS	404785091845...	11	17	13.35
Perl5	fed34a19f844...	10	6	7.50
Rails	bf2b9d2de3f8...	47	52	49.37
Voldemort	2a5c69145fb6...	4	8	5.33
Zotero	d456117ebe9c	3	5	3.75

branches 1 and 2. The harmonic mean is a plausible metric because it allows for finding scenarios with high values in both branches.

Consider, for example, the merge with most developers in the Rails project, with harmonic mean of 49.37, having 47 developers that committed on branch 1 and 52 developers that committed on branch 2. For this case, the complexity of identifying the key participants to solve a certain merge conflict can be rather high. The possibilities of communication exceed 2,000 channels among the developers involved. On the other hand, the Zotero project has harmonic mean of 3.75, leading to 15 different channels of communication among developers.

The developers that performed commits in a given branch can help in identifying which participant is more suitable to participate in the collaborative merge. That is, who has the potential to know the changes made in such a merge.

5.4. *Q4: How many merges have the same developers in both branches?*

The fourth question observes the number of merges having the same developers in both branches. Table 5 displays information on the intersections of developers in

Table 5. Intersection of developers in merging branches.

	Intersection of developers					
Projects name	Exactly the same (a)		Some intersection (b)		Without intersections (c)	
Django	1	0.25%	35	8.66%	368	91.09%
Git	435	5.85%	3926	52.82%	3073	41.33%
JQuery	19	7.60%	72	28.80%	159	63.60%
MaNGOS	7	2.74%	71	27.85%	177	69.41%
Perl5	15	0.93%	95	5.91%	1497	93.16%
Rails	56	0.91%	886	14.41%	5205	84.68%
Voldemort	64	15.76%	83	20.44%	259	63.80%
Zotero	34	15.25%	94	42.15%	95	42.60%

merging branches. The first column presents the project names. The remaining columns show the quantity and proportion of:

(a) Merges that have all the developers in common in the branches;
(b) Merges that have some developers in common; and
(c) Merges that have different developers in both branches.

As seen in the Django project, only 1 merge (0.25%) has exactly the same developers in both branches and 8.66% have the some intersection. In the Git project, for instance, the intersection reaches more than 50% of the merges, and the same developers participate in both branches in about 5% of the merges. In the JQuery, MaNGOS, Voldemort, and Zotero projects, the number of people in both branches is also expressive, with more than 30% of the merges having the some intersection. The Django, Perl, and Rails projects feature small intersections of developers in the branches.

The fact that there are developers who know both of the branches being merged can aid in the designation of key participants to take part in the collaborative merge. It is believed that, as the developer participating in the merge knows parts of the work that is being completed in both branches, the chances of performing a more coherent choice during conciliation increases.

6. Discussion about the Survey

This section presents which actions developers take when they need to merge source code, and especially when a conflict arises during the merge. The survey was performed from June 2014 to July 2014 and 164 developers from Brazil, USA, France, and Indonesia answered it. The answers for the 16 questions, divided into three sections, are shown in the following.

From the 164 participants, most of them are between 21 and 40 years old (61% are between 21 and 30 years old and 32% are between 31 and 40 years old). Most of them have BS (42%) or MS (32%) degrees. Some are undergraduate students (21%) and few have a PhD (5%) degree. Graphics about this information are shown in Fig. 4.

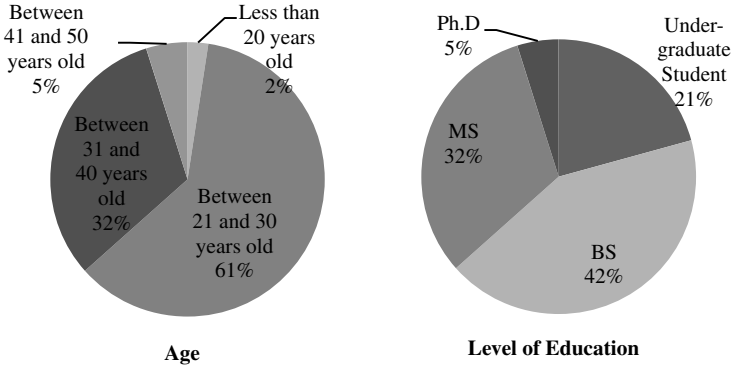

Fig. 4. Age and level of education.

Regarding professional experience, we could extract information such as the level of experience with version control systems in academic and/or industrial projects, the size of the development team, and the adopted development methodology. As depicted in Fig. 5, 45% of the developers have experience in using version control systems in academic and industrial projects, while 40% have experience only in industrial projects. Moreover, 3% of the developers have no experience at all and were discarded because they would not help in understanding how people manage conflicts during merge.

From the 139 developers with experience in industrial projects, many have more than 7 years of experience (44) and between 4 and 7 years of experience (46). On the other hand, from the 94 developers with experience in academic projects, most of them have between 1 and 4 years of experience (56) and between 4 and 7 years of experience (22). As expected, developers from the industry have the more experienced ones. The results are shown in Fig. 6.

Still regarding the matter of professional experience, we asked the participants about the average number of members in the development team that they had worked with. In this question, the developer was allowed to mark more than one

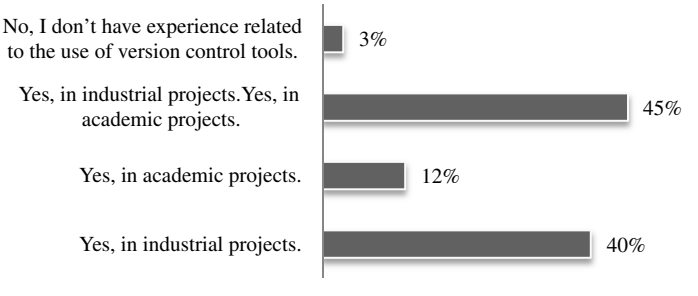

Fig. 5. Experience related to the use of version control tools.

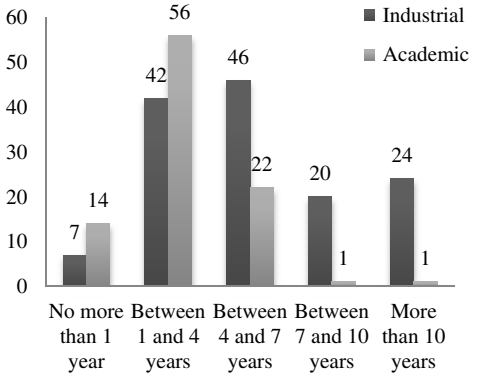

Fig. 6. Time dedicated to academic and industrial projects.

answer, and 124 developers marked that they worked in teams with up to 10 members, while 52 answered that they worked in teams from 11 to 20 members. Based on the result, we can conclude that most of the developers worked in small teams, which is a common strategy adopted by companies in recent years, mainly because of the adoption of agile methodologies.

Actually, this hypothesis holds, as most developers in fact are using agile methodologies (133). Moreover, 66 developers answered that they worked with incremental model and 46 developers worked with waterfall model. Among the 46 developers who cited the waterfall model, only 4 of them exclusively used the waterfall model. The other 42 also cited other models, especially agile. On the other hand, agile methodologies were exclusively cited by 47 people. The information is show in Fig. 7.

We also wanted to observe which tools developers used to adopt for version control. This helps us to understand the strategies they use for merging. The distribution of tools developers have experience with is the following: CVS (94),

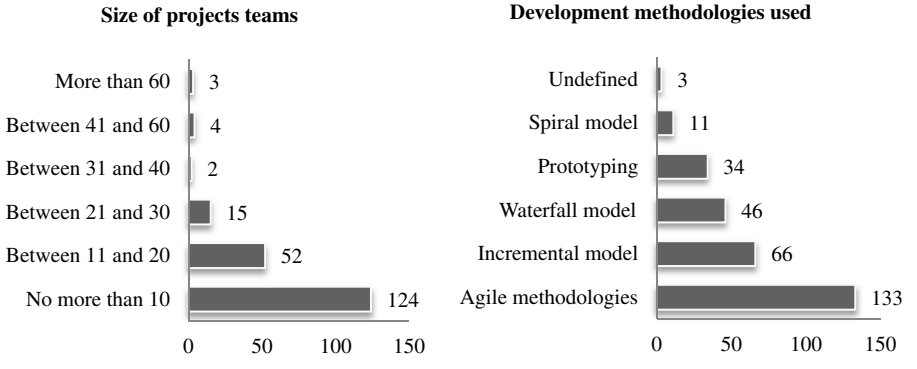

Fig. 7. Project team size and development methodologies used.

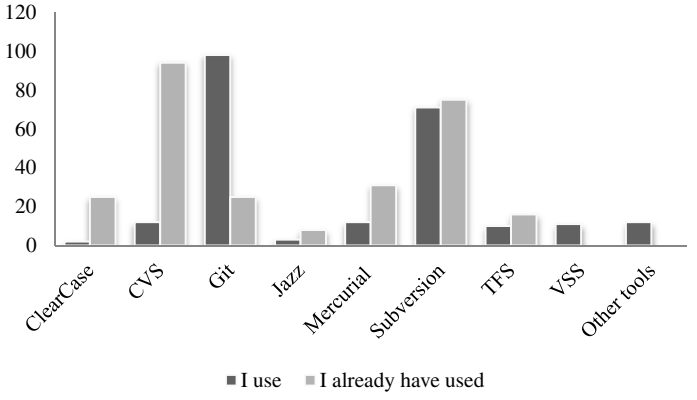

Fig. 8. Version control tools developers used (or use) in software projects.

Subversion (75), Mercurial (31), ClearCase (25), and Git (25). However, they are currently using the following systems: Git (98) and Subersion (71). Figure 8 depicts this information. It is important to highlight that CVS is the tool that most people have experience, but only 12 developers are currently using it. Another important piece of information is about Git tool.

After understanding the profile of the experiment participants, we probed deeper regarding how they perform merge. In the first question, we asked how often they try to commit their changes before other team members submit theirs in order to avoid merge. As shown in Fig. 9, most developers answered that they always (13%), frequently (28%), or sometimes (22%) try to commit before other developers. This shows that it is a common practice committing early to avoid merge conflicts. However, at some point someone will have to perform the merge and resolve possible conflicts anyway.

Two other questions addressed in the study were related to the policies adopted in the projects: (1) policies to avoid merge conflicts, and (2) policies to select developers to perform the merge. Figure 10 shows the results for both questions. It is possible to

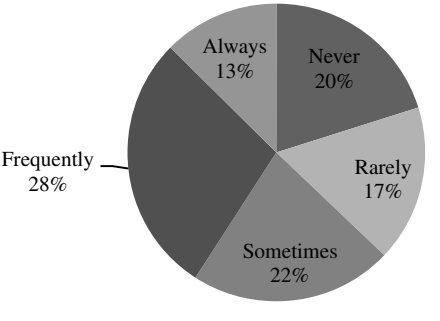

Fig. 9. How often developers try to commit their changes before other team members to avoid merge.

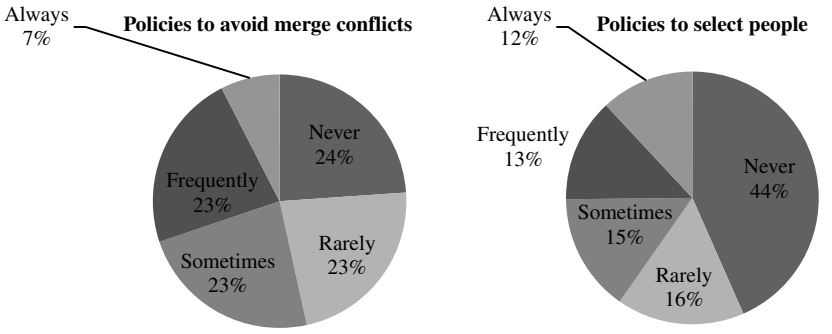

Fig. 10. Teams that adopt policies to avoid merge conflicts and adopted policies to select the person (or people) to merge branches.

observe that few developers always adopt some policies to avoid merge conflicts (8%). The developers also mentioned some policies that they used to adopt to avoid merge: (a) division of features per developer (23), (b) small and very frequent commits (13), (c) report the team before performing the merge (12), and (d) use of branches to each feature (10).

It is important to observe that some policies only postpone the need to merge, and indirectly make the merge more complex. For example, one participant cited that he uses the policies "a" and "b", and neither of them was infallible. It is interesting to notice that, although people who cited policy "a" have not clearly mentioned the use of branch, division of tasks is usually assisted by the use of branch, as reported in policy "d".

Regarding policies to choose the developer responsible to perform the merge, 43% of developers told that they never used any policy. Among developers that adopt policies, the most cited were: (a) the most experienced developer (15), (b) the configuration manager (10), (c) people involved in a conflict (6), and (d) the technical lead (4).

The developers were asked about how often they make decisions over source code that was coded by other developers. Many developers answered they sometimes (33%), frequently (23%), or always (5%) make this kind of choice. This shows that it is common to have developers dealing with source code "owned" by other developers during merge.

The developers also answered a question about how they feel when they need to make decisions alone over source code that was coded by other developers. Most of them said that they sometimes (30%), rarely (24%), or never (14%) feel comfortable to do it. Figure 11 summarizes this data.

The developers who feel confortable in making decisions over others' code argued that code is collective and developers must know all the source code. The developers who said they sometimes feel comfortable argued also that it depends on knowledge they have about the code. Those who said rarely or never feel comfortable also

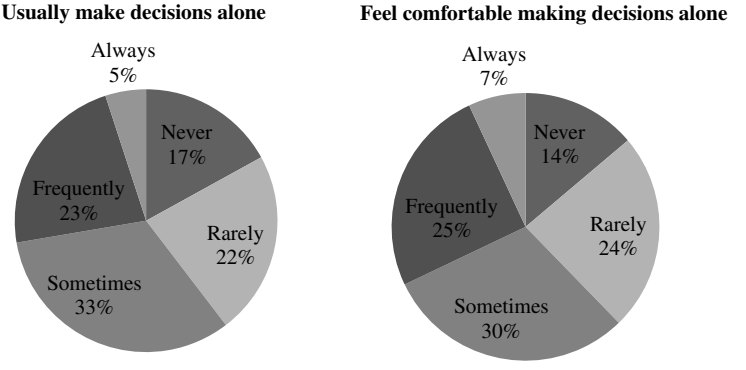

Fig. 11. Decisions made alone about changes in parts of the code e how developers feel about it.

argued that sometimes there is no coding standard, and the best way to avoid misunderstandings is to chat with the developer who "owns" the code.

Furthermore, we tried to understand how often the developers exchange information with other developers to solve merge conflicts (Fig. 12). Most of the developers said that they always (36%), frequently (39%), or sometimes (16%) ask for information to other developers. Thus, we can see that in most cases (75%) developers need assistance to resolve conflicts. Ideally, the developer who will aid must be the one who has more knowledge (e.g. edited most of the code) of the conflict.

The last question is about how the developers choose who is responsible to solve merge conflicts. Most of the developers said they frequently use the version control history to make this decision: 42% of developers always use the history, while 31% of them frequently use this history. Although most developers use the history information to choose the developer who will solve the conflict, 75% of them do not use automated tools in this task. Other developers make their decision based on the developer who made only the last change: 43% of the developers said that they use this approach frequently and 23% said they always use this approach. Finally, 45%

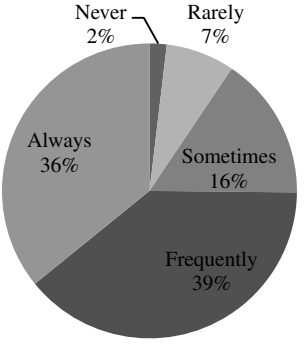

Fig. 12. Contacted other developers to resolve the conflict together.

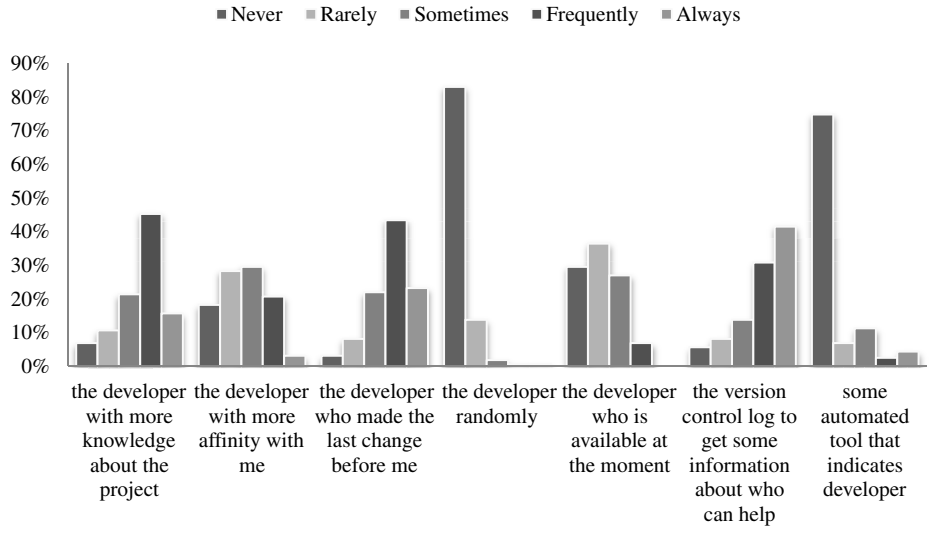

Fig. 13. How often the developers take into consideration when requesting help from other developers to resolve a merge conflict.

developers said they frequently indicate the developer who has more knowledge in the project. Finally, factors like affinity and availability were considered as relevant when a conflicting merge occurs. Figure 13 shows all the options available and the developers' choice.

7. Developers' Assignment

From findings described in Secs. 5 (post-hoc analysis) and 6 (survey), it is possible to devise a method to help the assignment of developers to the branch merging task. However, there is no obvious approach to solve this problem, because different developers use different techniques. Moreover, as projects are constantly evolving, this choice becomes more difficult to make manually, given the number of developers on each branch and the difficulty to quantitatively assess the contributions of each developer as well as their value.

We could identify some forces that may guide in devising such assignment method. From the post-hoc analysis we could observe that some developers perform more commits than others on each branch and some developers contribute to both branches. On the other hand, from the survey we could observed that experience is a key factor for choosing someone to merge branches.

In this section we use the knowledge obtained from the post-hoc analysis and the survey to indicate developers for merging branches in three real-world complex scenarios. We also show some heuristics based on the aforementioned forces that aim at better selecting developers for this task. It is important to notice that we use only the version control history to support our suggestion in the following subsection.

7.1. *Git*

From the project history analysis, the merge with the most complex setting of developers, by harmonic mean, has 158 developers in branch 1, and 10 developers in branch 2. Besides, 5 developers are committed in both branches. Table 6 shows the amount of commits made by each one of these 5 developers.

Table 6. Developers in both branches for Git.

	Developers in both branches — Git project		
Merge	Developers	Commits in branch 1	Commits in branch 2
a8816e7bab03...	Markus	23	5
	Alex	7	1
	Shawn	2	4
	Jens	1	4
	Ferry	1	1

For this merge, the average number of commits per developer is 8.9 in branch 1 and 2.3 in branch 2. In this case, although *Markus* was not the last developer to commit, should a conflict arise, he would be a strong candidate to participate in the collaborative merge, as he committed above the average in both branches.

Apart from the verification of developers who worked in both branches, it is possible to verify the participants who performed most commits. Table 7 shows the 6 developers who committed the most, 3 in the first branch and 3 in the second branch. Besides possibly knowing a lot about the work being completed, if there is an intersection between the two analyses, the weight of the indication of the participant must be higher. For this merge, there are some intersections between developers in both analyses. Therefore, *Junio*, *Markus*, *Shawn*, and *Jens* would be candidates to be part of the collaborative merge.

Table 7. Developers with most commits.

	Developers with the most commits — Git project		
Merge	Developers	Commits in branch 1	Commits in branch 2
a8816e7bab03...	Junio	**512**	0
	Schindelin	**69**	0
	Jeff	**58**	0
	Markus	23	**5**
	Shawn	2	**4**
	Jens	1	**4**

7.2. *Perl5*

From the project history analysis, the merge with the most complex scenario of developers, by harmonic mean, has 10 developers in branch 1 and 6 developers in branch 2. Only 1 developer participated in both branches, as seen in Table 8.

Table 8. Developers in both branches for Perl5.

	Developers in both branches — Perl5 project		
Merge	Developers	Commits in branch 1	Commits in branch 2
fed34a19f844...	Hans	2	1

Table 9. Developers with most commits.

	Developers with the most commits — Perl5 project		
Merge	Developers	Commits in branch 1	Commits in branch 2
fed34a19f844...	Gurusamy	**17**	0
	Hugo	**2**	0
	Tom	**2**	0
	Jarkko	0	**16**
	Hans	2	**1**
	Kurt	0	**1**

For this merge, only *Hans* has committed in both branches. For this reason, he is a strong candidate to participate in the collaborative merge. In addition, he was also one of the six developers who committed the most (Table 9). Two other strong candidates to participate in the merge are *Gurusamy* and *Jarkko*, who were responsible for a large number of commits in branches 1 and 2, respectively, as shown in Table 9.

7.3. *Voldemort*

From the project history analysis, the merge with the most complex setting of developers, by harmonic mean, has 4 developers in branch 1 and 8 developers in branch 2. In this case, 4 developers participated in both branches, as shown in Table 10.

The developers who committed the most in branch 1 also committed in branch 2, which makes all of them (especially *Bhusesh*, *Kirktrue*, and *Alex*) strong candidates to perform the collaborative merge. In this case, the developers who participate in both branches were also the ones who performed the most commits, reinforcing the evidence that they should participate in the collaborative merge.

Table 10. Developers in both branches for Voldemort.

	Developers with the most commits — Voldemort project		
Merge	Developers	Commits in branch 1	Commits in branch 2
2a5c69145fb6...	Bhusesh	33	11
	Kirktrue	29	10
	Alex	20	13
	Bbansal	4	2

8. Conclusion

As shown in the previous sections, it is possible to use the history to identify the potentially best developers to take part in the collaborative merge session. This can be done based on forces identified in this study as potential ways to assign participants to merge branches, mainly through the identification of developers who committed in both branches and the ones who committed the most in one of the branches. These forces were used in a study of three real projects in this work.

As a contribution, this study presented two analyses, a post-hoc and a survey analysis, trying to demonstrate it is possible to identify potential developers to perform merge branches collaboratively regarding repository history.

As a threat to validity, the post-hoc analysis was based on the committers' names and/or email addresses, which can generate inconsistency in the numbers presented. Furthermore, the analyses were carried out in merges that had already occurred. For this reason, it was impossible to effectively measure the effects of including the suggested developers in the merge session. Moreover, this study does not take into consideration developers' previous knowledge of libraries or programming languages being used, but solely the commit history, that is, the participation of the developers in the specific project under analysis.

In a future work, we hope to apply this research in pending merges, in order to observe the advantages of indicating who is the most suitable developer. Moreover, we intend to conceive a method for indicating the most suitable developers and automated such method. Finally, we want to add additional analysis related to the expertise over changed artifacts and the knowledge proximity amongst developers.

Acknowledgments

The authors would like to thank CAPES, CNPq, and FAPERJ for the financial support.

References

1. A. Nieminen, Real-time collaborative resolving of merge conflicts, presented at the *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing* (CollaborateCom), 2012, pp. 540–543.
2. A. Sarma, D. F. Redmiles and A. van der Hoek, Palantir: Early detection of development conflicts arising from parallel code changes, *IEEE Transactions on Software Engineering* **38**(4) (2012) 889–908.
3. J. Portillo-Rodriguez, A. Vizcaino, C. Ebert and M. Piattini, Tools to support global software development processes: A survey, presented at the *5th IEEE International Conference on Global Software Engineering* 2010, pp. 13–22.
4. H. C. Estler, M. Nordio, C. A. Furia and B. Meyer, Unifying Configuration Management with Merge Conflict Detection and Awareness Systems, Washington, DC, USA, 2013, pp. 201–210.

5. J. Young Bang, D. Popescu, G. Edwards, N. Medvidovic, N. Kulkarni, G. M. Rama and S. Padmanabhuni, CoDesign: A highly extensible collaborative software modeling framework, presented at the *ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 2, 2010, pp. 243–246.
6. Y. Brun, R. Holmes, M. D. Ernst and D. Notkin, Proactive detection of collaboration conflicts, 2011, pp. 168–178.
7. J. Wloka, B. Ryder, F. Tip and X. Ren, Safe-commit analysis to facilitate team software development, in *Proceedings of the 31st International Conference on Software Engineering*, Washington, DC, USA, 2009, pp. 507–517.
8. L. Hattori and M. Lanza, Syde: A tool for collaborative software development, presented at the *ACM/IEEE 32nd International Conference on Software Engineering*, 2010, Vol. 2, pp. 235–238.
9. M. L. Guimarães and A. R. Silva, Improving early detection of software merge conflicts, Piscataway, NJ, USA, 2012, pp. 342–352.
10. C. Costa and L. Murta, Version control in distributed software development: A systematic mapping study, presented at the *IEEE 8th International Conference on Global Software Engineering*, 2013, pp. 90–99.
11. C. Costa, J. J. C. Figueirêdo and L. Murta, Collaborative merge in distributed software development: Who should participate? in *26th International Conference on Software Engineering and Knowledge Engineering*, Vancouver, Canada, 2014, pp. 268–273.
12. T. Mens, A state-of-the-art survey on software merging, *IEEE Trans. Software Engineering* **28**(5) (2002) 449–462.
13. S. P. Berczuk and B. Appleton, *Software Configuration Management Patterns: Effective Teamwork and Practical Integration* (Addison-Wesley, Boston, Mass, 2003).
14. M. Koegel, H. Naughton, J. Helming and M. Herrmannsdoerfer, Collaborative model merging, New York, NY, USA, 2010, pp. 27–34.
15. S. L. Pfleeger and B. A. Kitchenham, Principles of survey research: Part 1: Turning lemons into lemonade, *SIGSOFT Softw. Eng. Notes* **26**(6) (2001) 16–18.
16. S. Chacon and J. C. Hamano, *Pro Git*, Vol. 288 (Apress, Berkeley, NY). Distributed to the book trade worldwide by Springer-Verlag, 2009.